STAT 133: Lab 6

Junyu Wang

### Conditional Expressions

If-else statements are perhaps the simplest type of control flow structure:

x <- 3 # Change this value!  
if (x > 0) {  
 print("positive")  
} else {  
 print("not positive")  
}

## [1] "positive"

Of course, the else is not necessary (if you don't want anything to occur otherwise), or you can chain multiple if-else statements:

y <- 10 # Change this value!  
if (y > 0) {  
 print("positive")  
} else if (y < 0) {  
 print("negative")  
} else {  
 print("zero?")  
}

## [1] "positive"

Your turn! Write R code that will "squish" a number into the interval [0, 100], so that a number less than 0 is replaced by 0 and a number greater than 100 is replaced by 100.

z <- 100\*pi  
# Fill in the following if-else statements. You may (or may not) have to add or subtract else if or else statements.  
if ( z < 0) { # Replace TRUE with a condition.  
 z <- 0  
} else if ( z > 100) { # Replace TRUE with a condition.  
 z <- 100  
}   
z

## [1] 100

If you find yourself using many if-else statements with identical structure for slightly different cases, you may want to consider a switch statement instead.

# Convert the day of the week into a number.  
day <- "Tuesday" # Change this value!  
switch(day, # The expression to be evaluated.  
 Sunday = 1,  
 Monday = 2,  
 Tuesday = 3,  
 Wednesday = 4,  
 Thursday = 5,  
 Friday = 6,  
 Saturday = 7,  
 NA) # an (optional) default value if there are no matches

## [1] 3

Switch statements can also accept integer arguments, which will act as indices to choose a corresponding element:

# Convert a number into a day of the week.  
day\_num <- 5 # Change this value!  
switch(day\_num,  
 "Sunday",  
 "Monday",  
 "Tuesday",  
 "Wednesday",  
 "Thursday",  
 "Friday",  
 "Saturday")

## [1] "Thursday"

Your turn again! Write a switch statement to determine if a given lowercase letter is a vowel. Assume that "y" is not a vowel.

letter <- "a" # Change this value!  
switch(letter,  
 e = ,  
 i = ,  
 o = ,  
 u = ,  
 a = "vowel"  
 ) # Modify as necessary.

## [1] "vowel"

### Loops

Loops are used when you want to perform a given task many times.

The simplest type of loop in R is the repeat loop, which will repeatedly evaluate a set of expressions until it is told to stop.

# Divide a number by 2 until it becomes odd.  
val\_rep <- 898128000 # Change this value!  
repeat {  
 print(val\_rep)  
 if (val\_rep %% 2 == 1) { # If val\_rep is odd,  
 break # end the loop.  
 }  
 val\_rep <- val\_rep / 2 # Divide val\_rep by 2 since val\_rep was even.  
 # When the end of the loop is reached, return to the beginning of the loop.  
}

## [1] 898128000  
## [1] 449064000  
## [1] 224532000  
## [1] 112266000  
## [1] 56133000  
## [1] 28066500  
## [1] 14033250  
## [1] 7016625

Often you will want to perform a loop until some condition is satisfied, or as long as a condition as satisfied. In that case, a while loop may be more appropriate.

val\_while <- 898128000 # Change this value!  
while (val\_while %% 2 == 0) { # Continue the loop as long as val\_while is even.  
 print(val\_while)  
 val\_while <- val\_while / 2  
}

## [1] 898128000  
## [1] 449064000  
## [1] 224532000  
## [1] 112266000  
## [1] 56133000  
## [1] 28066500  
## [1] 14033250

print(val\_while)

## [1] 7016625

Notice that the while loop code is more compact (ignoring the print statements), since the stopping condition has been encoded in the loop itself.

Your turn! Write R code which multiplies a positive number by 3 and adds 1 until the result is greater than 10000. For example, 2015 -> 6046 -> 18139. Write both a repeat loop and a while loop.

n\_rep <- 2020 # Change this value!  
repeat {  
 n\_rep <- 1 + n\_rep \* 3  
 if (n\_rep > 10000) break # Replace this with your code.  
}  
n\_while <- 2020 # Change this value!  
while (n\_while < 10000) { # Replace FALSE with your stopping condition.  
 # Fill in.  
 n\_while <- 1 + n\_while \* 3  
}

Sometimes instead of performing an action until a condition is satisfied, you just want to perform it a specified number of times. In these cases, a for loop is appropriate.

for (i in 1:10) { # Repeat 10 times.  
 print("Hello world!")  
}

## [1] "Hello world!"  
## [1] "Hello world!"  
## [1] "Hello world!"  
## [1] "Hello world!"  
## [1] "Hello world!"  
## [1] "Hello world!"  
## [1] "Hello world!"  
## [1] "Hello world!"  
## [1] "Hello world!"  
## [1] "Hello world!"

Your turn again! Write a for loop to add 1 to every element of a vector.

vec <- c(3, 1, 4) # Change this value!  
for (i in 1:length(vec)) { # Replace c() with an appropriate sequence.  
 # Fill in.  
 vec[i] <- vec[i] + 1  
}

### Basic Functions, with Control Flow Structures

Often you will find that it is easier to begin writing code for a specific task, with specific variables, before generalizing the code into a function. Earlier in the lab, we saw the following code, which divides a positive integer by 2 until it becomes odd.

x <- 898128000 # Change this value!  
while (x %% 2 == 0) {  
 print(x)  
 x <- x / 2  
}

## [1] 898128000  
## [1] 449064000  
## [1] 224532000  
## [1] 112266000  
## [1] 56133000  
## [1] 28066500  
## [1] 14033250

print(x)

## [1] 7016625

Your turn! Now generalize the above code to create a function which performs the same operation. (You should change very little.)

reduce <- function(x) {  
 # Fill in.  
 while (x %% 2 == 0) {  
 print(x)  
 x <- x / 2  
 }  
}  
  
reduce(898128000)

## [1] 898128000  
## [1] 449064000  
## [1] 224532000  
## [1] 112266000  
## [1] 56133000  
## [1] 28066500  
## [1] 14033250

How about something a little bit more complicated? Write a function f which, given a positive integer n, returns n / 2 if n was even or 3 \* n + 1 if n was odd.

f <- function(n) {  
 # Fill in.  
 if (n %% 2 == 0)   
 n <- n / 2  
 else   
 n <- 3 \* n + 1  
 n  
}  
  
f(6)

## [1] 3

f(19)

## [1] 58

f(27)

## [1] 82

Now create function g which applies f() to a positive integer n until it becomes 1, and returns the number of iterations until this occurs.

g <- function(n) {  
 # Fill in.  
 x = 0  
 while (n != 1){  
 n <- f(n)  
 x <- x + 1  
 }  
 i  
}  
  
g(6)

## [1] 3

g(19)

## [1] 3

g(27)

## [1] 3

If you didn't earlier, try also writing the same function, but now create f() as a nested function.

h <- function(n) {  
 f <- function(m) {  
 # Fill in.  
 counter <- 0  
 while (m != 1){  
 if (m %% 2 == 0)   
 m <- m / 2  
 else   
 m <- 1 + 3 \* m  
 counter <- 1 + counter  
 }  
 i  
 }  
 # Fill in.  
 n <- f(n)  
 n  
}  
  
h(6)

## [1] 3

h(19)

## [1] 3

h(27)

## [1] 3

### Default Arguments

As you probably know by now, many functions come with default arguments, which can be left unspecified without affecting the execution of the function.

The following function attempts to compute the length of the hypotenuse of a right triangle, but will accept one or two leg lengths.

pythagoras <- function(a, b = a) {  
 return(sqrt(a^2 + b^2))  
}  
  
pythagoras(3, 4)

## [1] 5

pythagoras(5)

## [1] 7.071068

**Your turn!** Write a function which adds together the elements of a vector raised to a certain power. Use a loop instead of vectorized operations. Include two arguments: vec and pow. Give pow a default value of 2.

# Remove eval = FALSE or set eval = TRUE when you have completed this function.  
powersum <- function(v, p = 2) { # Fill in arguments.  
 # Fill in.  
 sum <- 0  
 for (i in v) {  
 sum <- sum + i ^ p  
 }  
 sum  
}  
  
powersum(1:5) # Should return 55.

## [1] 55

powersum(1:5, 4) # Should return 979.

## [1] 979

### Ellipsis (...)

Finally, in addition to normal and default arguments, the ... argument allows a function to take any number of arguments.

The following function simply counts the number of arguments given.

countargs <- function(...) {  
 print(paste("This function received", length(list(...)), "arguments."))  
}  
  
countargs(0)

## [1] "This function received 1 arguments."

countargs(1L, 5, 3 + 5i, TRUE, "yosemite")

## [1] "This function received 5 arguments."

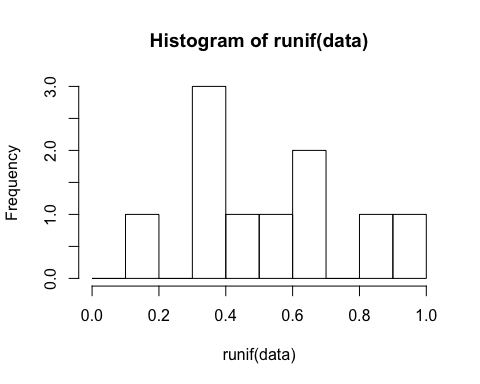
countargs(c(1, 2, 3, 4, 5), factor(c("one", "two", "three")))

## [1] "This function received 2 arguments."

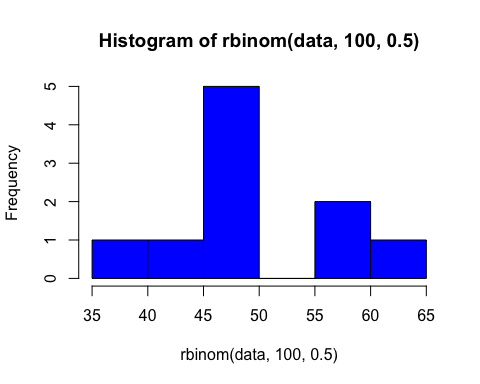
More commonly, however, the ... is simply passed as optional arguments to another function.

**Your turn!** Write a function which produces a histogram of a random process. The first argument should be n, the number of observations. The second argument should be one of the following: "Uniform", "Binomial", "Poisson". These should correspond, respectively, to: runif(n), rbinom(n, 100, 0.5), rpois(n, 1). Finally, include the ... argument to be passed into the histogram function.

# Remove eval = FALSE or set eval = TRUE when you have completed this function.  
randhist <- function(data, dis,...) { # Fill in arguments.  
 # fill in  
 switch(dis,  
 "Uniform" = hist(runif(data), ...),  
 "Binomial" = hist(rbinom(data, 100, 0.5), ...),  
 "Poisson" = hist(rpois(data, 1), ...)  
 )  
}  
  
randhist(10, "Uniform", breaks = seq(0, 1, 0.1))



randhist(10, "Binomial", col = "blue")



randhist(10, "Poisson")

